Comp 11 Lectures

Mike Shah

Tufts University

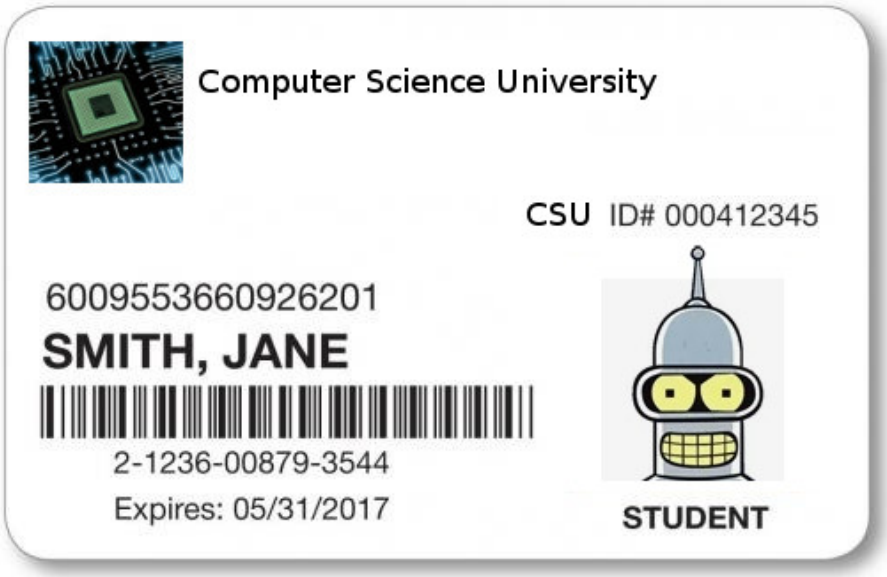
June 20, 2017

Please do not distribute or host these slides without prior permission.

Structs

Comp 11 - Pre-Class warm up

How many different data types do you see on the student ID below. (e.g. Computer Science University is a string)



Lecture

Edsgar Dijkstra

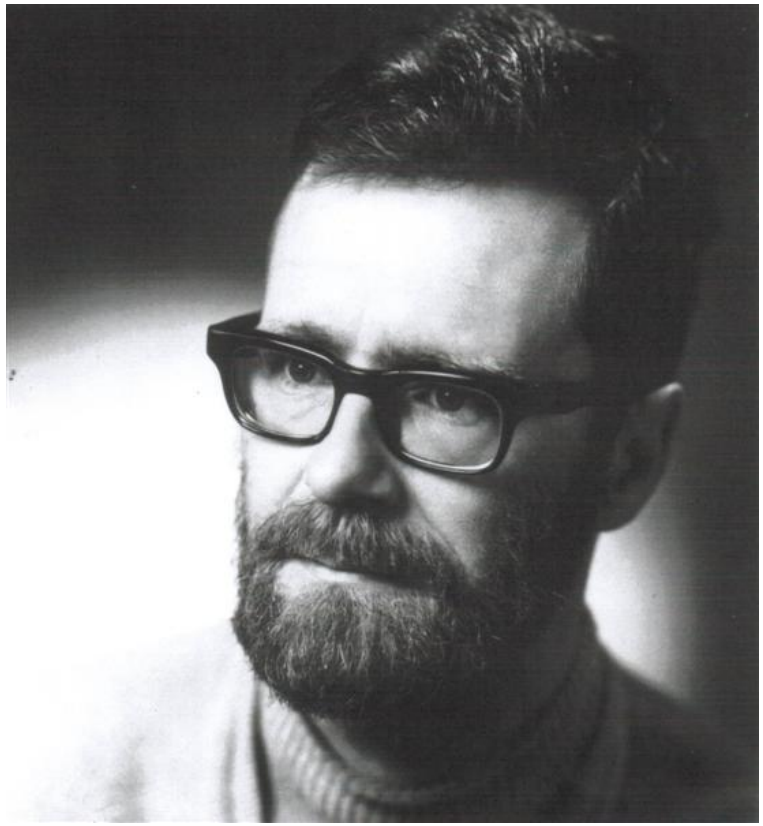
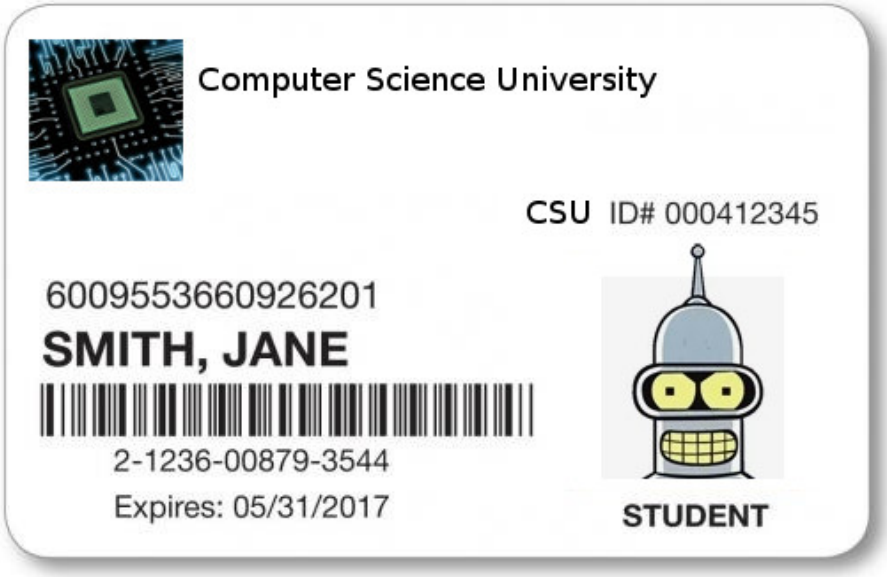


Figure 1: Edsgar Dijkstra was one of the pioneering computer scientists forming many foundational algorithms and ideas in computer science. His work on concurrency and distributed computing is what makes things like the cloud possible. Variations of Dijkstra's shortest path algorithm are what make your GPS pick a path for you to get from location A to B.

Data Types

- The student name, a string
- An ID, perhaps an int or long
- A leading digit on the barcode, perhaps store as a char or int.
- A date (3 ints, month, day, year)
- And several more pieces



Program a Student ID

So lets actually write the program for storing a student ID.

```
#include <iostream>
#include <string>

int main(){

  // Student Data
     std::string name = "Smith, Jane";
     long id = 000412345;
     char digit = '2';
     std::string date = "5/31/17";

  return 0;
}
```

Listing 1: A program that takes in a student ID

How does our program scale?

By now, you know the next series of questions:

- What if I want to add another student?
- What abstraction might I use to help?

Building a student database

How about an array? Or even a std::vector

```
#include <iostream>
#include <string>
#include <vector>

int main(){
    std::vector<std::string> nameVector;
    std::vector<long> idVector;
    std::vector<char> digitVector;
    std::vector<std::string> dateVector;

    nameVector.push_back("Smith, Jane");
    idVector.push_back(000412345);
    digitVector.push_back('2');
    dateVector.push_back("5/31/17");

  return 0;
}
```

Listing 2: Now we can build a database of students

Populating our Student Database

```
1  // Add our first student
2      nameVector.push_back("Smith, Jane");
3      idVector.push_back(000412345);
4      digitVector.push_back('2');
5      dateVector.push_back("5/31/17");
6  // Add a second student
7      nameVector.push_back("Einstein, Albert");
8      idVector.push_back(1042323);
9      digitVector.push_back('2');
10     dateVector.push_back("5/31/17");
11 // Add a third student
12     nameVector.push_back("Granger, Hermione");
13     idVector.push_back(9993235);
14     digitVector.push_back('3');
15     dateVector.push_back("5/31/17");
```

Listing 3: Now we can build a database of students

Strategy to Update database

Now what if a students information changes? How would we update our database.

- Well, we know we can access the student by an index.
- And that index will be the same for each field
- That is, accessing nameVector[0], idVector[0], etc. holds our first students information in each vector.

Accessing a student

```
1 // Lets update our third student, Hermionie
2     nameVector[2] = "Granger, Hermione Jean";
3     idVector[2] = 0;
4     digitVector[2] = '7';
5     dateVector[2] = "1/31/18";
```

Listing 4: If we need to update our student

Does everyone like this solution?

- First, does this work and update Hermione's student information in each of the 4 vectors?
- What do you think about this approach in general?

Figure 2: Hermione Granger is a fictional character from the Harry Potter book series.

Potential problems

Sometimes we forget things!

```
// Oops -- I forgot 1 field
    nameVector.push_back("Smith, Jane");
    digitVector.push_back('2');
    dateVector.push_back("5/31/17");
// Oops -- I forgot 2 fields!
    nameVector.push_back("Einstein, Albert");
    dateVector.push_back("5/31/17");
// Add a third student
    nameVector.push_back("Granger, Hermione");
    idVector.push_back(9993235);
    digitVector.push_back('3');
    dateVector.push_back("5/31/17");
```

Listing 5: I forgot to push_back all the fields in each of our vectors. Now Ms. Granger is not always index number 2

Code Length

- We have to remember to work with 4 vectors all of the time.
- What if we want to pass this information to a function? We have to have four parameters that pass the correct data type
- What if I want to update my student record with a fifth field? We need to find every place in the code and update it.
- This leaves us prone to errors.

Our next abstraction

- The point is, we want an abstraction to help make our lives easier
- A way to *struct*ure related pieces of data together.
- Luckily, C++ (And most languages) have a tool to do this.

The struct

struct

- A **struct** is a C++ keyword
- It allows us to create a new data type that we define.
- This data type can consist of multiple primitives.
- All of the data stays together.
- Lets take a look

Our first struct

```
1 // We now have a type called Student (with a capital S)
2 struct Student{
3   // Our 4 fields of data
4     std::string name;
5     long id;
6     char digit;
7     std::string date;
8 };
9 // Known ad a POD - piece of data
```

Listing 6: We have created a new data type. It is our own custom data type that we have defined for our programs.

Our struct in C++

```
#include <iostream>

// We define our struct outside of main
struct Student{
    // Our 4 fields of data
    std::string name;
    long id;
    char digit;
    std::string date;
};

int main(){
    // We simply use the name of our struct as
    // the new datatype
  Student janeSmith;

  return 0;
}
```

Listing 7: Defining our struct

Setting Data in our struct

```
#include <iostream>
struct Student{
    std::string name;
    long id;
    char digit;
    std::string date;
};

int main(){
    Student janeSmith;
    // Quite similar to our very first example!
    janeSmith.name = "Smith, Jane";
    janeSmith.id = 000412345;
    janeSmith.digit = '2';
    janeSmith.date = "5/31/17";

  return 0;
}
```

Listing 8: Note that the . operator is how we access our fields

Summarizing what is new and important terminology

- We create a struct before our main, which allows us to use a new data type.
- Our struct can have several data types, that when aggregated make another type. .
- The . (DOT) operator allows us to access and update each of a struct's individual fields.
- We can think of a struct as a large piece of data (POD) that we define. We like to think of these as objects rather than variables (which are our primitive types like int, float, char, bool, etc).
- We call Student, an 'object' type. So janeSmith is an *instance of an object* called Student

Adding more power to struct

- A struct has fields (member fields we call them) such as int, char, string, as previously seen.
- But we can also have member functions, that can be called on our objects.
- Lets create a member function, that outputs all of our fields values.

Our output member function in use

```
1  struct Student{
2    std::string name;
3    long id;
4    char digit;
5    std::string date;
6    // We create a regular function
7    // The difference is, it is within our struct.
8    // So this function can only be used on Student's
9    void output(){
10     std::cout << name << "\n";
11     std::cout << id << "\n";
12     std::cout << digit << "\n";
13     std::cout << date << "\n";
14   }
15 };
```

Listing 9: Defining our struct

output usage (warning text is tiny)

caption

```
#include <iostream>

struct Student{
    std::string name;
    long id;
    char digit;
    std::string date;

  void output(){
        std::cout << name << "\n";
        std::cout << id << "\n";
        std::cout << digit << "\n";
        std::cout << date << "\n";
    }
};

int main(){
    Student janeSmith;
    janeSmith.name = "Smith, Jane";
    janeSmith.id = 000412345;
    janeSmith.digit = '2';
    janeSmith.date = "5/31/17";
    // Call our output method on janeSmith.
    // Remember janeSmith's datatype is of Student, so we can do this
    janeSmith.output();

    return 0;
}
```

Is this better than what we started with?

```
1 janeSmith.name = "Smith, Jane";
2 janeSmith.id = 000412345;
3 janeSmith.digit = '2';
4 janeSmith.date = "5/31/17";
```

Listing 10: Here we define individual fields one at a time

- This bit of the code is a bit ugly.
- How could we get rid of it?

add a set member function

```
1  struct Student{
2      std::string name;
3      long id;
4      char digit;
5      std::string date;
6      // Our new set method
7      void set(std::string _name, long _id, char _digit, std::
       string _date){
8        name = _name;
9          id = _id;
10         digit = _digit;
11         date = _date;
12     }
13
14   void output(){
15         std::cout << name << "\n";
16         std::cout << id << "\n";
17         std::cout << digit << "\n";
18         std::cout << date << "\n";
19     }
20 };
```

What is our set member function doing?

- We created a new member function called set that takes four parameters
- We then assign whatever values are passed into that function to our member variables. This sets the member variables by a simple function call.
- Remember, our member variables are defined within our struct, but outside of our function–thus they are within scope. That is why they persist.

Using set

```
1 int main(){
2   Student janeSmith;
3   // Ah, much better. Now only one line of code
4   janeSmith.set("Smith, Jane", janeSmith.id, 000412345, '2',
      "5/31/17");
5   // We output all of our freshly set member variables
6   janeSmith.output();
7
8   return 0;
9 }
```

Listing 12: We call the member function set on janeSmith

Creating multiple students

```
1  int main(){
2    Student janeSmith;
3    janeSmith.set("Smith, Jane", 000412345, '2', "5/31/17");
4    janeSmith.output();
5
6    Student albertEinstein;
7    albertEinstein.set("Einstein, Albert", 1042323, '2', "
       5/31/17");
8    albertEinstein.output();
9
10   Student hermioneGranger;
11   hermioneGranger.set("Granger, Hermione", 9993235, '3', "
       5/31/17");
12   hermioneGranger.output();
13
14   return 0;
15 }
```

Listing 13: We create more students and they each have their own member variables unique to themselves

Full Example (Try on your own after class)

```
#include <iostream>

struct Student{
    std::string name;
    long id;
    char digit;
    std::string date;

void set(std::string _name, long _id, char _digit, std::string _date){
      name = _name;
        id = _id;
        digit = _digit;
        date = _date;
    }

  void output(){
        std::cout << name << "\n";
        std::cout << id << "\n";
        std::cout << digit << "\n";
        std::cout << date << "\n";
    }
};

int main(){
  Student janeSmith;
  janeSmith.set("Smith, Jane", 000412345, '2', "5/31/17");
  janeSmith.output();

  Student albertEinstein;
  albertEinstein.set("Einstein, Albert", 1042323, '2', "5/31/17");
  albertEinstein.output();

  Student hermioneGranger;
  hermioneGranger.set("Granger, Hermione", 9993235, '3', "5/31/17");
  hermioneGranger.output();

  return 0;
}
```

C++ everywhere!

Or try it here! (Press run and see output at the bottom of the page)
`https://wandbox.org/nojs/clang-4.0.0/permlink/coUYabq0LMniR9WG`

Final Takeaways

- We have discovered a new abstraction that allows us to move beyond primitive data structions.
- We can create and use member functions (In the same way we have seen member functions with string).
- We can also combine this abstraction with others we have learned.
  - Passing a Student struct in a function
  - Creating a vector of our Student struct is now possible.
- Hmm, maybe we should try the above in lab.

Review of what we learned

- (At least) Two students
- Tell me each 1 thing you learned or found interesting in lecture.

No in-class activity today

5-10 minute break and then...

To the lab!

Lab: `http://www.mshah.io/comp/11/labs/lab6/lab.pdf`
[1]

[1] You should have gotten an e-mail and hopefully setup an account at `https://www.eecs.tufts.edu/~accounts` prior to today. If not–no worries, we'll take care of it during lab!